NTNU – Trondheim
Norwegian University of
Science and Technology

Enhancing Software Portability with Hardware Parametrized Autotuning

Henrik Holenbakken
Knutsen

Master of Science in Computer Science
Submission date: October 2013
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Jan Christian Meyer, NTNU-IT

Norwegian University of Science and Technology
Department of Computer and Information Science

Enhancing Software Portability with Hardware Parametrized Autotuning

Henrik Holenbakken Knutsen
Department of Computer and Information Science
Norwegian University of Science and Technology

Master of Science in Computer Science
Submission date: September 2013
Supervisor: Dr. Anne C. Elster
Co-supervisor: Jan Christian Meyer

Problem Description

This thesis will investigate and evaluate the possibilities of using autotuning tools to optimize scientific applications based on the hardware architecture. Orio is the autotuning toolkit that will be used, and the codebase that will be investigated is Code_Saturne, a general purpose computational fluid dynamics software package.

The goal is to perform an autotuning process and evaluate the results to describe if and how autotuning tools can be used to enhance software portability and performance. Ideally, the autotuning can be implemented in such a way it will result in run-times in the range of, or better than those of generalized algebra libraries such as ATLAS for any architecture.

Assignment given: 06. May 2013
Supervisor: Anne Cathrine Elster, IDI
Co-advisor: Jan Christian Meyer, NTNU-IT

Abstract

In recent years multicore computing has taken the step from being new technology to being used everwhere. Todays supercomputers now have thousands of cores, and even smartphones use multicore processors. This technological advance has been an important factor in development of scientific applications.

A major challenge in the development of these applications is performance, a problem further increased by the variety of used platform architectures. To achieve near maximum performance on a specific architecture is a tedious and difficult process of manual optimization. Furthermore, such optimizations often have little to no effect on other architectures.

This challenge has encouraged the use of libraries offering highly optimized mathematical functions. Autotuning software is an attractive alternative for general and good optimization, for that reason some of these libraries also take advantage of autotuning. Recently, autotuning toolkits such as Orio have become available. These tools derive information about the system architecture, and use it in conjunction with user-specified parameters to alter source code to the best possible fit for the current architecture.

In this project Code_Saturne has been explored with the use of profile analysis tools to find core code sections with respect to run-time. Autotuning directives have been implemented in these functions to achieve architecture specific implementations of said functions.

Instrumentation and profiling identified central functions, which were explored as candidates for optimization. However, autotuning prove to be feasible for only a small number of visits to these functions. The autotuning problem was very dependent on the type and size of input, which reduced the impact of auto-tuning to the point where it could not be reliably observed. This suggests it is required to integrate the tuning process with the application.

Sammendrag

I de siste årene har bruk av flerkjerners prosessorer tatt steget fra å være ny teknologi til å bli brukt overalt. Dagens superdatamaskiner har tusenvis av kjerner, og selv smart-telefoner har flerkjerners prosessorer. Denne teknologiske utviklingen av vært en viktig faktor i utviklingen av vitenskapelige applikasjoner.

En stor utfordring innen utvikling av slike applikasjon er ytelse, ett problem som gjøres enda vanskeligere av alle forskjellige maskin-arkitekturer som er tilgjengelig. Å oppnå maksimum ytelse på en spesifikk arkitektur er en krevende og vanskelig manuell prosess. Og slike optimaliseringer har ofte liten effekt på andre arkitekturer.

Denne utfordringen har gjort bruk av bibliotek som tilbyr veldig optimaliserte matematiske funksjoner populære. Autotuning verktøy er attraktive alternativer for gode og generelle optimaliseringer, derfor tas autotuning i bruk også av mange av disse bibliotekene. Nylig har autotuning verktøy som Orio blitt tilgjengelige. Disse verktøyene bruker informasjon om systemets arkitektur sammen med parametre brukeren har spesifisert til å endre kode til å passe en arkitektur best mulig.

I dette prosjektet ble Code_Saturne utforsket ved bruk av verktøy for profileringsanalyse, for å finne seksjoner av kode som er sentrale i forhold til kjøretid. Autotuning direktiver har så blitt implementert i disse funksjonene for å oppnå arkitektur-spesifikke implementasjoner av disse funksjonene.

Instrumentasjon og profilering oppdaget sentrale funksjoner som ble utforsket og evaluert som kandidater for optimalisering. Det viste seg at det kun var mulig å optimalisere et fåtall av kallene til disse funksjonene. Autotuning-problemet var veldig avhenging av type og størrelse på input, noe som reduserte effekten av autotuning så mye at den ikke kunne observeres med sikkerhet. Dette indikerer at autotuning prosessen må integreres med applikasjonen.

Acknowledgements

This thesis was completed at the High Performance Computing Group at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU) in collaboration with the HPC-section of the IT-department at NTNU.
I would like to thank my supervisor Dr. Anne C. Elster for giving me the chance to work at the HPC-Lab over the last year, offering interesting projects. Her guidance and knowledge of the field has also been a great help in completing this thesis. I would also like to thank Bjørn Lindi and Dr. Jan Christian Meyer for giving me the chance to work on a masters thesis related to the Partnership for Advanced Computing in Europe (PRACE) project. Dr. Jan Christian Meyer deserve extra gratitude for his efforts in helping to resolve problems with using the Vilje supercomputer. I would also like to thank the other students at the HPC-lab for creating an enjoyable working environment. Finally, I want to thank NTNU and NVIDIA for their support of and donations to the HPC-lab.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers have been an important factor in science, being able to solve complex and massive numerical simulations at great speed. This has mainly been the task of the CPU, but over the recent years general-purpose computing on graphics processing units (GPGPU) has seen a steady increase [3].

Initially, it was the desire for gaming graphics that was the main driver for GPU development. However, the highly parallel and computational capabilities of the modern graphics cards are a good match to the numerous and simple mathematical calculations that are the basis of many scientific models[4]. This has led to GPUs emerging as a powerful platform for high-performance computation. Examples of GPGPU are use in applications such as a snow simulation [5] or for fluid dynamics [6].

A big problem with GPGPU is the discrepancy of the architectures of various graphics cards. Code optimized for one architecture will not necessarily be as efficient on another architecture, and in some cases the impact is great. This reduces the gain of the often tedious optimization process for a specific architecture, as these changes introduce limitations to what hardware that can be used if maximum performance is desired. A way to overcome this limitation is to make use of auto-tuning.

An auto-tuning system is capable of adapting source code based on configuration parameters and the environment, in order to achieve optimal performance for that architecture. This process can then be repeated on another architecture with very little to no modification, achieving optimal perfor-

mance there as well.

This thesis aims to explore the use of auto-tuning through the use of the Orio framework[1] to increase the performance of central algorithms scientific applications. Code_Saturne is going to be autotuned, it is a general purpose computational fluid dynamics package[7]. More details about Orio and Code_Saturne will be presented in Chapter 4.

The rest of the thesis is structured as follows:
This thesis is structured as follows:
Chapter 2 describes the recent development of parallel computing, and common architectures.
Chapter 3 provides background information about methods of auto-tuning.
Chapter 5 depicts the profiling analysis used to find central code segments.
Chapter 6 holds information about how the autotuning modifications are implemented in the source code.
Chapter 7 presents the results, offering a comparison of the test results.
Chapter 8 is dedicated to conclusions and discussion about future work.

Chapter 2

Parallel computing models

A recent trend in high performance computing is the use of graphics processing units for general purpose programming[3][8][9]. GPUs are highly parallel computational units, and the low cost and energy efficiency makes GPUs a very good alternative to CPUs for compute intensive tasks. This chapter will give a short introduction to parallel computing in general. General-purpose computing on graphics processing units (GPGPU) will also be explained.

2.1 Parallel computing

This section will introduce the basic concepts of parallel computing. More detailed information can be found in textbooks such as [10]. Further information with regards to parallel scaling an memory I/O limitations can be found in [11].

2.1.1 Parallel computer architectures

Shared memory vs distributed memory
In a single processor system, there is a single processor using a single, and often cached memory. As more processors are added two options arise: either give each processors its own memory - a distributed scheme, or let all the processors work with the same memory, a shared scheme. The two schemes are visualized in Figure 2.1 on the next page.

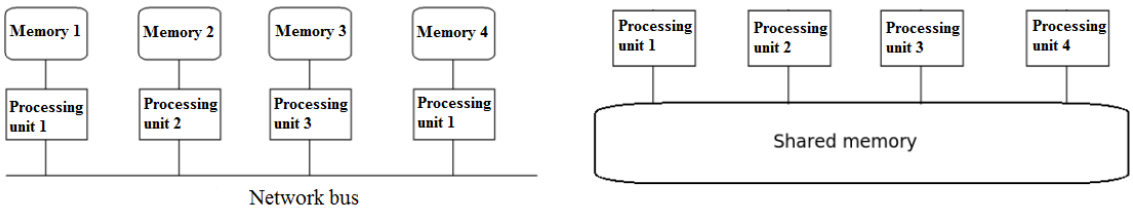


Figure 2.1: Overview of a distributed memory system (left) and a shared memory system (right).

In a distributed memory system, the nodes (processor + memory) must in some way be connected so they can communicate. With this scheme, a processor can not access the memory of other nodes. This means that to make it possible for several nodes to work on the same data they have to explicitly send data to other nodes through a network.

In this regard, the shared memory system is simpler. Here all the processors access the same, single memory. So for two processors to work on the same data, the second processor simply has to read the memory address(es) to which the first processor wrote.

A combination of the two schemes is also possible. For such a system each processing unit have both some private memory, as well as direct access to memory shared between all the processing units.

SIMD vs MIMD

The other classification is based on the type of instructions and data streams, also known as Flynn's taxonomy[12]. Non-parallel computers are single instruction stream, single data stream (SISD). The single processor is executing one instruction stream, and working on one set of data.

SIMD is the abbreviation for single instruction stream, multiple data streams. In this case there are multiple processors that each work on a different set of data, all performing the same instructions. This type of system is very useful when there are large amounts of data that need to be processed in the same way.

The alternative, multiple instruction, multiple data (MIMD) is the most flexible. Here each of the processors work on its own data set, possibly executing different instructions. Multiple instruction, single data (MISD) is the last of the four. This is a type of parallel computing architecture where processors perform different operation on the same data. The classification is summarized in Table 2.1

	Single instruction	**Multiple instruction**
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Table 2.1: Overview of Flynn's Taxonomy

2.1.2 Parallel Scaling

Increasing the number of processors would ideally decrease computational time by the same factor. However, very few problems are perfectly parallelizable, meaning more detailed estimates are required. Amdahls law[13] and its reevaluation known as Gustafson's law[14] are central in obtaining optimal multi-core performance. They can also be used to find the maximum overall performance increase for a program where only a part of the system is parallelized. Speedup S is given by:

$$S(N) = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{(1-p)+\frac{p}{N}}$$

where s is the fraction of execution time spent on the serial part, p=*1-s* is the proportion of the program that can be parallelized, and N is the number of processors. It can be deduced from this formula that for small p, the overall speedup will be small even for big N. This implies that optimizations should be directed at the parts of the program where most of the time is spent. For simple programs, these sections are often easy to find, *e.g.* nested for loops, but in complex cases the use of a profiler tool is a better choice.

When N grows to infinity,the maximum speedup is given by *1/(1-p)*. As p decreases the denominator approaches 1, meaning the value of adding more processors falls rapidly. As an example, if p is 0,9 (that is, 90% of the program can be parallelized) the maximum speedup is a factor of 10. As a result of this, parallel computing is the most effective for a small number of processors, or for problems with very high values of p - also called embarassingly parallel problems. This also introduces a new problem to efficient parallel computing: the task of designing the code so that the component *(1-p)* is as small as possible.

An estimate of p can be found using the measured speedup *(SU)* on the number of processors *(NP)* using the formula

$$P_{estimated} = \frac{\frac{1}{SU} - 1}{\frac{1}{NP} - 1}$$

This estimated p can then be used in Amdahl's law to predict the speedup of a program for different numbers of processors.

If *s'* and *p'* is the time spent on the serial and parallel parts on a parallel computer, the speedup will be:

$$S(N) = \frac{s' + p \cdot N}{s' + p'} - s' + p' \cdot N$$

The result is known as Gustafson's law. The two laws are equivalent, but based on different assumptions. Amdahl assumes that as the computer becomes more parallel, it will still be used to run the same problem, in other words time is constant. Gustafson assumes that the more parallel computer will be used on a larger instance of the problem, *i. e.* constant parallel time.

More information about performance modeling on heterogeneous systems can be found in [15] and [16]. They detail the trade-off between the complexity and accuracy of performance models, and the challenges created by utilizing these models for design decisions.

Chapter 3

Autotuning

This chapter will give an introduction to the topic of autotuning. In particular, Section 2.1 will explain what it is, Section 2.2 elaborates advantages and disadvantages of autotuning, while Section 2.3 gives an overview of the methodology.

3.1 Introduction

Both size and complexity of scientific computations are increasing at least as fast as improvements in processor technology. Programming of these scientific applications is often difficult, and optimizing them at a sufficient level even harder. The desire for both bigger and more precise models has resulted in a quest for high performance technology, leading to rapid changes and increasingly complex and specialized computing architectures.

A consequence of this is a potentially big gap between the current and peak performance of applications. Many applications have a performance as low as 10% or less of the peak [17]. Another concern is the inability of current languages, compilers and systems to deliver the available performance for the application through fully automated code optimizers.

With regards to scientific computing, it is important to achieve performance without degrading productivity. When developers try to improve the performance of scientific code, they generally attempt one or more of the following approaches: manual optimization of code fragments; use of libraries for key

numerical algorithms; or use of compiler-based source transformation tools for loop-level optimizations.

Manual tuning is both time consuming and will often severely degrade readability, portability and scalability. Tuned libraries are often an easy method to achieve great performance, but the functionality and portability offered by these libraries are often limited. The latter method, source transformation tools, have not yet gained popularity among computational scientists, mainly because of poor portability and a steep learning curve.

3.2 Advantages and disadvantages

Autotuning can be a simpler optimization method than the aforementioned techniques. An example is loop unrolling, which can either be managed by the compiler or done manually. However, autotuning can often find even better ways to unroll, especially when empirical testing is used to evaluate changes.

It is also often easy to implement, and requires little to no knowledge about the target architecture. In a simple case, all that is required is addition of a few directives in the source code to be optimized. The autotuning tool will then use the available information to generate a tuned version of the target code.

Another advantage is how small the imprint of the directives are. It is only an addition to currently existing code, not a replacement, so a version of the original code is maintained. That way, there is only a slight decrease in readability because of optimization. Finally, the main feature of autotuning is the increase in portability, achieved by doing the hardware parametrized optimization.

However, the autotuning process does require some knowledge about the source code. Tuning all of the source code is very inefficient, and this would also be a tedious process for very large projects even if it is easy to implement. The autotuning should therefore be directed at core functions. These can in some cases be obvious, in others they might require either first-hand

knowledge of the code, study of manuals, or profile analysis.

The scalability in terms of size of the source code is questionable. As mentioned, it is not only necessary to find core functions, but large projects with complicated compile and linking processes can complicate the procedure as the tuning process uses a wrapper compiler. It is not always as easy as just invoking the autotuning toolkits compiler on the code section that is to be modified.

Finally, the codes core functions must be in a form (*e.g.* loops) that can easily be translated to optimized code. As an example, if the majority of the run-time is spent in MPI calls, autotuning the code will have very little to no effect. However, most scientific code is of the form where core functions are simple mathematical operations.

3.3 Methodology

General autotuning involves three major phases:

1. Identifying code optimization techniques that are relevant to the given source code and architecture of the system

2. Assigning a range of parameter values using hardware expertise and application-specific knowledge

3. Search the parameter space to find the best performing configuration of parameters for the given architecture

Step one is to choose which optimization techniques that should be used by the autotuning toolkit. This is dependent on both the source code and the hardware architecture of the CPU or GPU. The next step is to assign a parameter space that is to explored. This can either be specific, narrowed by use of knowledge about hardware- and software-specific features, or very general only limited by the time it takes to search the entire parameter space. Finally, the autotuning tool searches the parameter space performing tests for each combination of parameters.

For each of these tests the run-time is compared to the best run-time to check if the current combination of parameter values is the optimum configuration so far, if it is not it will calculate how near it is to being the best. Some procedures also support further searching the area near a found optimum, thus enabling a more detailed search using smaller parameter steps in the area close to that where the initial optimum configuration was found. This attempt might lead to finding a slightly better configuration that was skipped because of too large steps in the initial search.

This recursive search is an elegant solution to the core problem of parameter determined autotuning, but it often requires considerable time to find the optimum configuration. There is a balance between the level of detail of the parameter search and the time it takes to run the target code for all combinations wof these parameter values. In some cases it can be necessary to limit the range or detail of the search, but this limitation can make the optimum configuration unachievable. However, recent work [18] shows that search problems arising from autotuning can be formulated as mathematical optimization problems, and illustrates the potential for mathematical optimization algorithms to find high-performing tuning parameters in a shorter computation time.

The parallel capability of graphics processing units are often utilized to achieve a significant speedup compared to CPU implementations. Many scientific applications also have multiple core algorithms executing simultaneously. In such applications each core function (kernel) should be autotuned independently, as they may each require a specific configuration to achieve the best performance. This is especially important as the impact of moving an application between GPU architecture is often bigger than moving between CPU architectures. The base of scientific applications are often simple mathematical equations, and their algorithms are very similar on GPUs and CPUs. For this reason a great variety of available optimization techniques and configurations are also available for the GPU platform, many of which also used for autotuning on CPUs.

Balaprakash *et. al.*[19] describes a set of extensible and portable Search Problems in Automatic Performance Tuning (SPAPT), aiming to aid in the development and improvement of search strategies and performance-improving transformations. SPAPT also contains representative implementations from

a number of lower-level, serial performance tuning tasks in scientific applications. Modeling, search space characteristics, and performance objectives are discussed. Finally, an illustrative experimental study is also presented in the paper.

Chapter 4

Orio, Scalasca and Code_Saturne

This chapter will introduce the toolkits and software package that will be used. Code_Saturne is the software package that will be investigated. Scalasca will be used for profiling analysis, and Orio for the autotuning process.

4.1 Orio

Orio is an extensible framework for annotated transformation and autotuning of codes written in different source and target languages, including transformations from a number of simple languages to C, Fortran and CUDA targets[1]. It generates code based on set configuration parameters, and empirically evaluates the performance of these generated code segments, ultimately selecting the one with the best performance.

There are several ways in which Orios annotation approach differs from existing compiler- and annotation-based systems. One is that by designing an extensible annotation parsing architecture there is no committing to a single general-purpose language. As a result of this annotation grammars which restrict the original syntax can be defined. This enables more effective performance transformations.

Another feature is that Orio was designed based on the following requirements: portability (which precludes extensive dependencies on external packages), extensibility (adding new functionality should require little or no change to the existing Orio implementation), and automation (Orio should provide tools that manage all steps of the performance tuning process).

Lastly, Orio is usable in real scientific applications without requiring reimplementation. This ensures that the significant investment in the development of scientific code is leveraged to the greatest extent possible. A high-level overview of Orio's code generation and tuning process is depicted in Figure 4.1 below.

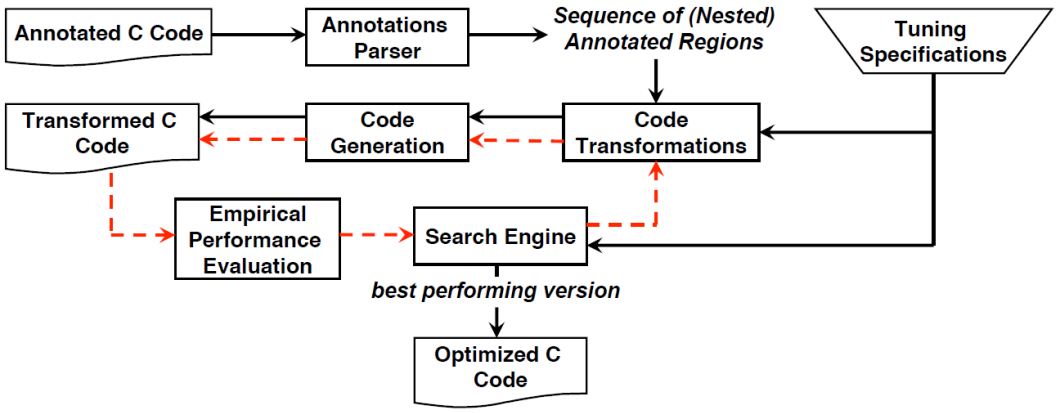


Figure 4.1: Overview of Orio's code generation and empirical tuning process. Figure from [1].

4.1.1 Related work

John Mellor-Crummey *et.al.* [20] describes LoopTool, which also supports annotation-based loop fusion, unroll/jamming, skewing and tiling. Qing Yi *et.al.* [21] presents POET: Parametrized Optimizations for Empirical Tuning. Azamat Mametjanov *et. al.* [22] have done work on autotuning stencil-based computations on GPUs, using the Orio framework. A autotuning framework for stencil computations on parallel multi-core has been described

by Shoaib Kamil *et. al.* in [23]. Active Harmony, I-Hsin Chung and Jeffrey K. Hollingsworth [24], is an automated runtime performance tuning system, intended to be used for large-scale scientific programs.

4.1.2 Annotation Language Syntax

Orio annotation is denoted as a stylized C comment starting with /*@ and ending with @*/. An annotation region consists of three main parts: leader annotation, annotation body and trailer annotation. In the form of a grammar, the structure of Orio annotations can be described as:

```
<annotation-region> ::=<leader-annotation><annotation-body><trailer-annotation>
<leader-annotation> ::=/*@ begin <module-name> (<module-body>) @*/
<trailer-annotation>::=/*@ end @*/
```

The annotation body can either be empty or contain C code that may include other nested annotation regions. The leader annotation holds the module name of the code transformation component that is to be used. A high level abstraction of the computation and the performance hints are coded in the module body inside the leader annotation and are used as input by the transformation module during the transformation and code generation phases. The annotated region is closed by the trailer annotation, which has a fixed form. Orio has two main functions: a *source-to-source transformation tool*, and an *automatic performance tuning tool*

4.1.3 As a source-to-source code transformation tool

There are several code transformation modules that are implemented and ready to use. One of them is the module for *loop unrolling*. This is a loop optimization aiming to increase register reuse and reduce branching instructions by combining instructions that are executed in multiple loop iterations into a single iteration. On the next page is an example of an annotated region set to use the loop unrolling module, with the unroll factor set to four.

```
/*@ begin Loop (
     transform Unroll(ufactor=4)
     for (i=0; i<=N-1; i++)
       y[i] = y[i] + a1*x1[i];
) @*/
for (i=0; i<=N-1; i++)
    y[i] = y[i] + a1*x1[i];
/*@ end @*/
```

The resulting unrolled code comprises two loops, one with the fully unrolled body and another loop for remaining iterations not included in the unrolled loop. In addition, the original code is included and available for use through setting a preprocessor variable.

4.1.4 As an automatic performance tool

When used as an automated performance optimizer, Orio adaptively generates a large number of code candidates with various parameter values for the given section of code, followed by execution of all these code variants and an empirical evalution. The code variant with the best performance is chosen, and its correlating parameters used. This entire process is automated through annotations: an example is given on the next page.

One PerfTuning module holds all the tuning parameters. This can also be done in a separate file to increase readability, and imported by the *PerfTuning* module. The other section holds the code that should be optimized, as well as the optimization techniques that will be used. The original code section can also be included so that the program can run normally without changing any files.

```
/*@ begin PerfTuning (
  def build {
    arg build_command = 'gcc -O3';
  }
  def performance_params {
    param UF[] = range(1,33);
  }
  def input_params {
    param N[] = [10,100,1000];
  }
  def input_vars {
    decl static double y[N] = 0;
    decl double a1 = random;
    decl static double x1[N] = random;
  }
) @*/
int i;
/*@ begin Loop (
  transform Unroll(ufactor=UF)
  for (i=0; i<=N-1; i++)
    y[i] = y[i] + a1*x1[i];
) @*/
for (i=0; i<=N-1; i++)
  y[i] = y[i] + a1*x1[i];
/*@ end @*/
/*@ end @*/
```

In this example the goal of the tuning process is to determine the optimal value of the unroll factor for different problem size. The *PerfTuning* module is used to define the tuning specifications that include the following four basic definitions:

- *build:* to specify all information needed for compiling and executing the optimized code
- *performance_params:* to specify values of parameters used in the program transformations
- *input_params:* to specify sizes of the input problem

- *input_vars:* to specify both the declarations and the initializations of the input variables

For this example, the code is compiled using the GCC compiler with the -O3 optimization option. The unroll factor is tested for values including integers from 1 to 32, inclusively. Three problem sizes are tested: N=10, N=100 and N=1000. Finally, all scalars and arrays used in the computations are declared and initialized in the tuning specifications, to enable the performance testing driver to empirically execute the optimized code.

4.2 Scalable Performance Analysis of Large-Scale Applications

Scalable Performance Analysis of Large-Scale Applications (Scalasca) is a software tool supporting performance optimization of parallel programs by measuring and analyzing their run-time behaviour [25]. It is specifically designed for large-scale parallel applications, attempting to aid in the problem of writing efficient code for modern supercomputers, which can have tens of thousand cores.

Applications run at this scale are often limited by excessive communication and synchronization overheads. This is especially true in cases where domains are irregular or dynamic, which causes wait states under message passing and computational imbalance if processes fail to reach synchronization points simultaneously. While covering single-node performance via hardware-counter measurements is possible, Scalasca is mainly targeted towards communication and synchronization issues as these are the most critical to achieve optimal performance levels of applications in the petaflops scale [26].

We will be using Scalasca to perform a profile analysis on the Code_Saturne package, to be able to extract important functions with regards to run-time. Even though Scalasca's main focus is MPI communication, it can also be used to find call central callpaths.

4.2.1 The Scalasca Architecture

The target application must be instrumented before any performance data can be collected, *i. e.* probes must be inserted into the code to trigger and carry out the measurements. This can be done at three different levels, source code, object code or through libraries. A Scalasca command is prepended to compile and link commands to activate instrumentation, the type of instrumentation is decided through arguments to the Scalasca command.

By default, MPI and OpenMP operations are instrumented, most compilers can also instrument all routines found in source files. Manual instrumentation can substitute automatic instrumentation and improve the structure of analysis reports, making them more comprehensible. Annotations are used to mark potentially nested sequences or blocks of statements, *e.g.* functions or loops. Measurements are controlled by a set of variables which can be specified in a configuration file in the working directory, or by setting the corresponding environment variables. Scalascas architecture is depicted in Figure 4.2 below.

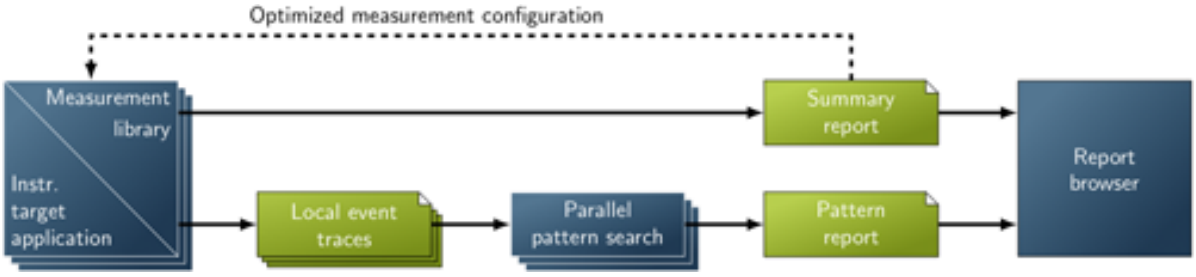


Figure 4.2: The Scalasca architecture. Figure courtesy of scalasca.org

The user can choose between generating a runtime summary report, or an event trace before running the instrumented executable. When tracing is enabled, each process generates a trace file containing records for its process-local events. It is recommended to optimize a instrumentation based on a previously generated summary report to avoid the instrumentation producing too large or inaccurate traces.

After application termination, Scalasca loads trace files into main memory and analyzes them in parallel, using as many cores as was used for running the application. Wait states are located and classified by category, and quantified by their significance. The final result is a wait-state report similar in structure to the summary report, but enriched with higher-level communication and synchronization inefficiency metrics.

Both summary and wait-state reports contain performance metrics for every combination of function call path and process/thread, and can be interactively examined in the provided analysis report explorer along the dimensions performance metric, call tree, and system. Reports can also be combined and/or manipulated for comparisons, aggregations or extracts of reports. Examples are comparing two reports to check the effect of an optimization, or remove uninteresting phases like initialization.

4.3 Code Saturne

Code_Saturne is a software package whose general purpose is computational fluid dynamics [27]. Development started in 1997 at Électricité de Frances R&D, and it is distributed under the GNU GPL license. It solves the Navier-Stokes equations for 2D, 2D-axisymmetric and 3D flows, steady or unsteady, laminar or turbulent, incompressible or weakly dilatable, isothermal or not, with scalar transport if required.

A variety of turbulence models are available, from Reynolds-Averaged models to Large-Eddy Simulation models. In addition, several more specific physical models are also available as modules: radiative heat transfer, fuel combustion, magneto-hydro gas, compressible flows, two-phase flows, Joule effect, electric arcs, weakly compressible flows, atmospheric flows and rotor/stator interaction for hydraulic machines.

It is based on the Finite-Volume method [28][29]. This method can easily be formulated to allow for unstructured meshes, as a result of this meshes with any type of cell (tetrahedral, hexahedral, pyramidal etc) and any type of grid structure (unstructured, block, hybrid etc.) is accepted. Code_Saturne is composed of two main elements and an optional GUI, as show in figure 4.3:

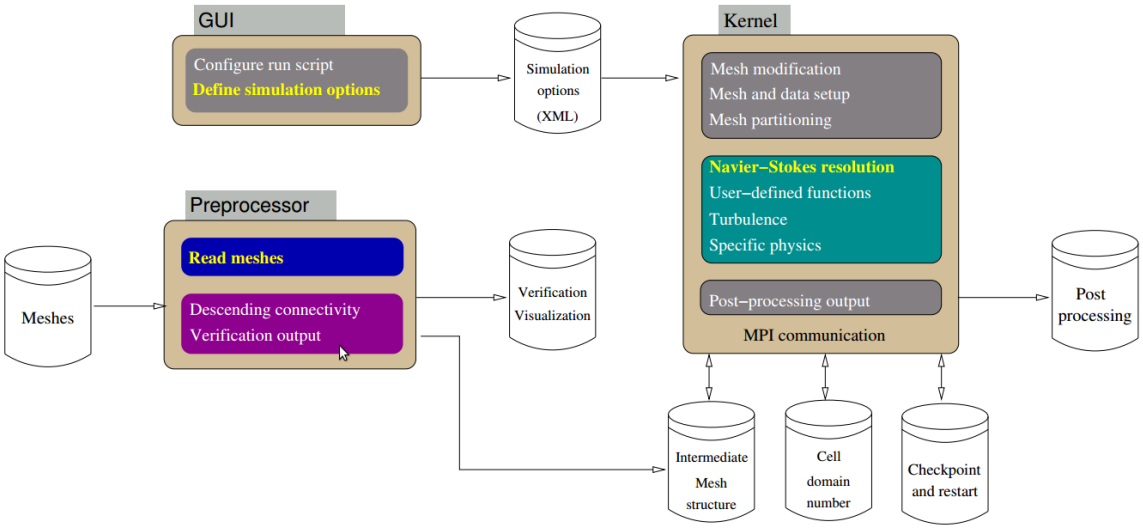


Figure 4.3: Core elements of Code_Saturne. Figure from [2].

The preprocessor handles reading and verficiation of chosen meshes, as well as handling connectivity between meshes in the case several are used. The preprocessor does not handle partitioning of meshes in the case where MPI is used.

This task is part of the kernel, where mesh modification, data setup and partitioning is done. The solver finds the solution to the Navier-Strokes equations, either through the use of built-in or user-specified funtions. Post-processing and output writing is also handled by the Kernel. MPI communication is executed by the kernel for every step.

The graphical user interface offers easy navigation of the many simulation options. Inserting all the parameters using the command line can easily become incomprehensible. Especially the mesh setup and configuration is made easy by the use of the GUI. Information about the navigation of meshes and how they are connected will quickly get complicated if done through command line. The simulation options are saved in XML notation, so it is also possible to import a complete set of simulation options. Simulation results can be read directly by visualization tools such as Paraview or EnSight.

Chapter 5

Profiling instrumentation method and results

This chapter will describe the profiling of Code_Saturne. The entire process is covered: how instrumentation of Code_Saturne was set up, the computation cases that were profiled and their results. Two Code_Saturne cases will be tested, one being a tutorial example running on a desktop computer, the other sizable enough to require a supercomputer.

5.1 Setting up instrumentation

Scalasca is used for the profiling analysis of Code_Saturne. The profile analysis is run on the Vilje[30] supercomputer at NTNU, where the Scalasca modulefiles are of version 1.4.2. This module must be loaded before Code_Saturne is configured, so that it is included in the setup. During configuration build folders are created, and their makefiles generated based on the system state and environment, as well as input arguments to the configuration.

Every folder and subfolder has its own Makefile. Instrumentation with Scalasca is done in two steps, an additional Scalasca argument must be prepended to both the original compile and execute commands. For more details, see section 4.2. In the case of running a large project on a supercomputer it is a bit more complicated. With regards to profiling analysis only the source folder is of interest, as it is this folder that contains the files in which instrumentation needs to be inserted.

Each of the makefiles consist of thousands of lines, however it is only one that needs to be changed to enable instrumentation. That is the variable indicating which C compiler that should be used, *CC*. As Code_Saturne was configured to use intel compilers for instrumentation, this variable was originally set to:

```
CC = icc
```

This must be extended to include the Scalasca command, so it is changed to:

```
CC = scalasca -instrument icc
```

All the Makefiles in the source folder were modified by prepending this Scalasca command. Makefiles were then invoked so that all the C files in the source folder were compiled with Scalasca instrumentation. Output from the makefile generation should indicate that files are compiled with scalasca in addition to icc.

Step two is to once again prepend a Scalasca command, this time to the original execution command. This is difficult to do directly because of how Code_Saturne is using python wrapper scripts to invoke execution. Another problem is that Vilje uses SGI MPT[31] as the MPI library, which is not included in the list of MPI libraries Code_Saturne can autodetect. Thus, another workaround is needed to be able to successfully execute using MPT. The section responsible for assembly of the MPI execution command is found in the file cs_case.py. The relevant lines (960-964) are given below.

```
mpi_cmd = ''
mpi_cmd_exe = ''
mpi_cmd_args = ''
if n_procs > 1 and mpi_env.mpiexec != None:
    # ... omitted ...
```

The first line is simply changed to:

```
mpi_cmd = 'scalasca -analyze mpiexec_mpt '
```

This change hardcodes Code_Saturne to use the MPI execution command defined and used by SGI MPT. The mpi_cmd string is later inserted into the execution command created by the python wrapper. With the source files now instrumented during compilation, and the wrapper set up to correctly use MPT, a Code_Saturne case can be run to create a profile analysis.

5.2 Profiling results

This section will cover the results of the profiling analysis. The main focus will be to pinpoint the function(s) where the majority of the run-time is spent. Only the computationally intensive functions are of interest. Functions managing MPI communication will also be encountered as it is responsible for a large portion of the run-time. However, they will be disregarded as this is code that can not be optimized with the use of autotuning.

Two cases has been profiled: One a PRACE UEABS large benchmark problem, sizable enough to be executed on a supercomputer. The other a small example from Code_Saturne's tutorial, designed to run on a desktop computer. As they both resulted in a similar call-graphs, and distribution of time spent in various functions only results from the PRACE case will be presented for the sake of simplicity.

As the Scalasca profile will hold all the functions called during all steps of execution it is unfeasible to present call-graphs in a perceptible way using screencaptures of Scalascas GUI. For that reason call-paths will be visualized using figures only including the relevant functions, *i. e.* those responsible for the majority of the run-time. Of the functions called by the main method it is *cs_run* that takes up almost all of the run-time. Other than that the main function only calls a handful very small setup functions, as well as the MPI initialization.

Figure 5.1 on the next page shows the important functions called by *cs_run*.

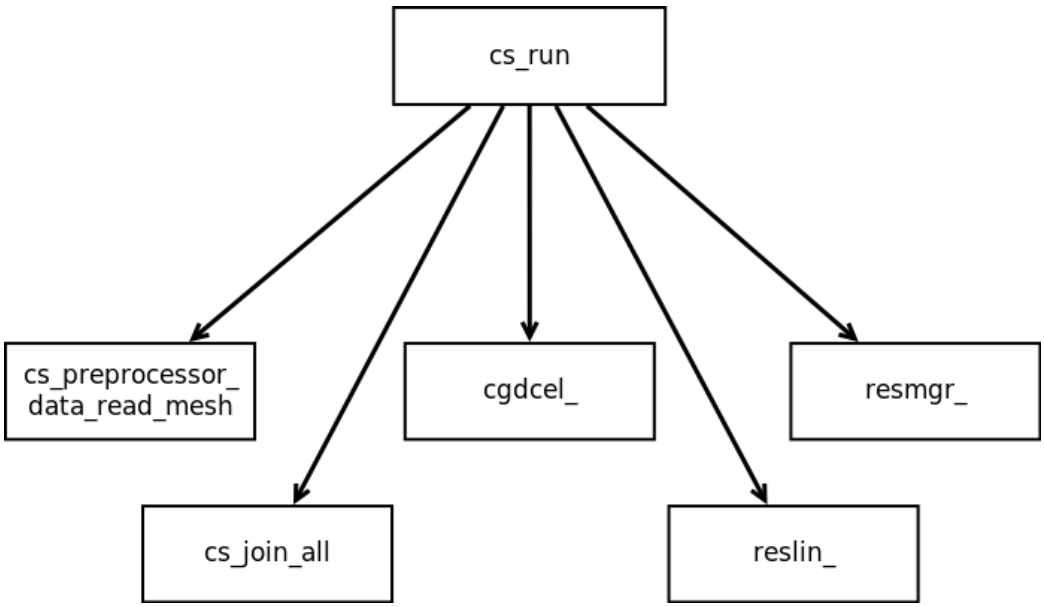


Figure 5.1: The important functions called by the cs_run function.

Added together, these functions and their subsequent calls are responsible for 96.54% of the total run-time. These are all massive functions, with lots of subroutines. The two leftmost functions are part of the preprocessor module of Code_Saturne, while the other three are part of the main kernel.

The remainder of this section will dive further into the callgraphs for each of these functions to pinpoint their base functions, which are responsible for the majority of the run-time. The functions found with this method will be evaluated as possible targets for an autotuning process in Section 5.3.

cs_preprocessor_data_read_mesh
This function is part of the preprocessing module, and handles reading and basic partitioning of the input data mesh. It has three major base functions. Figure 5.2 below covers the call graph for this function. MPI is not used in the preprocessing module, so this function is purely computational.

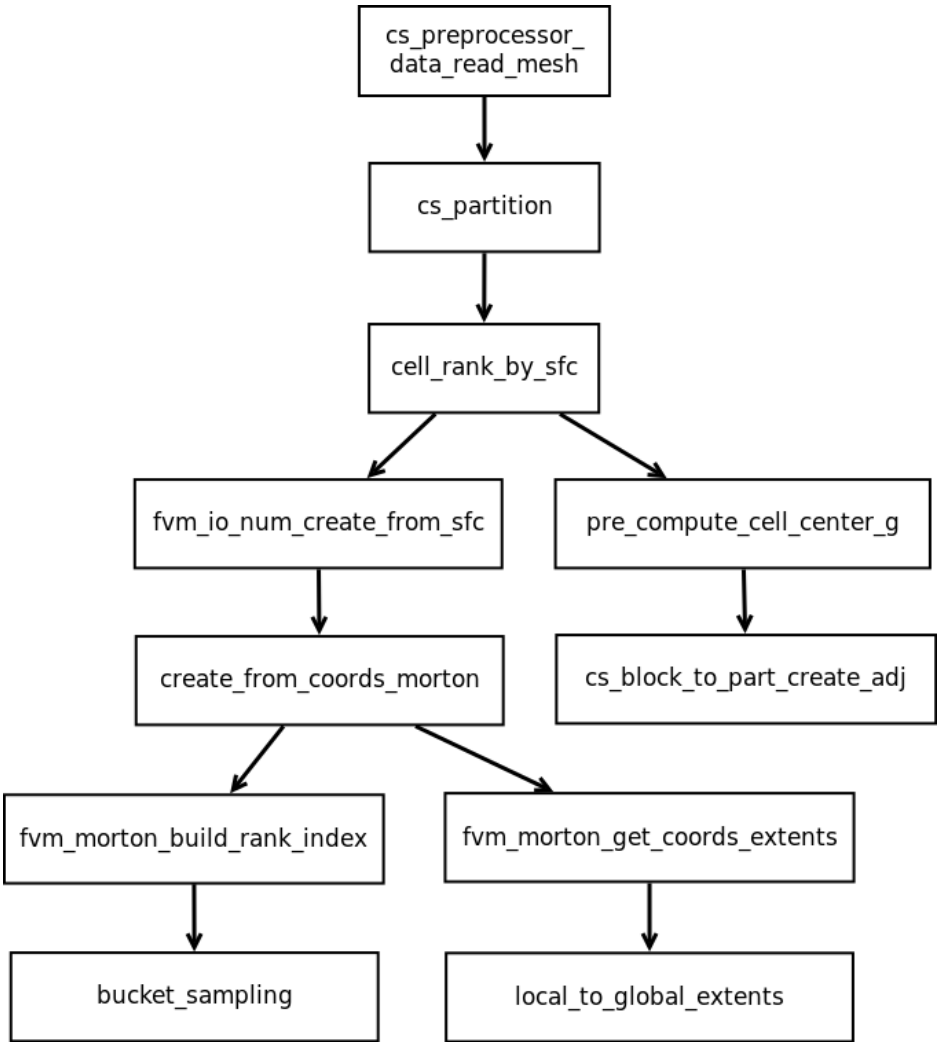


Figure 5.2: Call graph for the cs_preprocessor_data_read_mesh function.

cs_join_all

This function is also part of the preprocessing module. It is responsible for applying the defined join operations on selected meshes. Figure 5.3 below gives an overview of the call graph for this function. As this is also part of preprocessing, no MPI communication is invoked here.

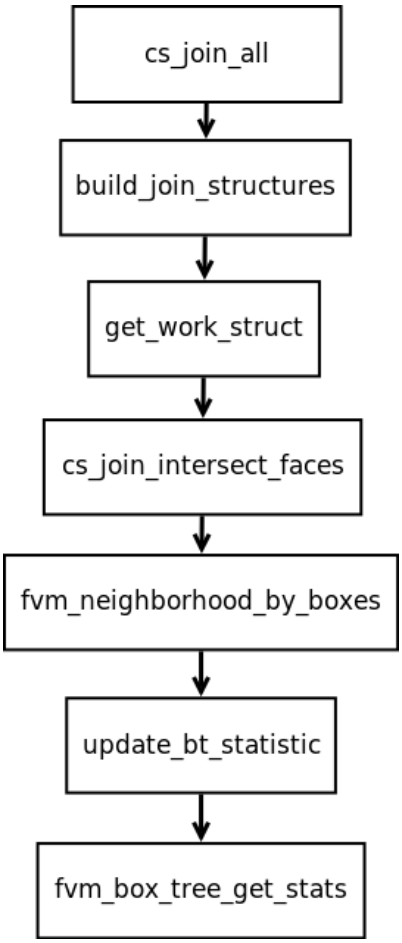


Figure 5.3: Call graph for the cs_join_all function.

cgdcel_

This is the first function to be part of the computational kernel of Code_Saturne. It is the gradient calculation that is performed by this function. It is a two step process, first the scalar gradient is intialized, then the iterative calculation is executed. MPI communication is performed after each iteration, and can be found in the cs_halo_syn_var_strided function. Figure 5.4 below holds the call graph for this function.

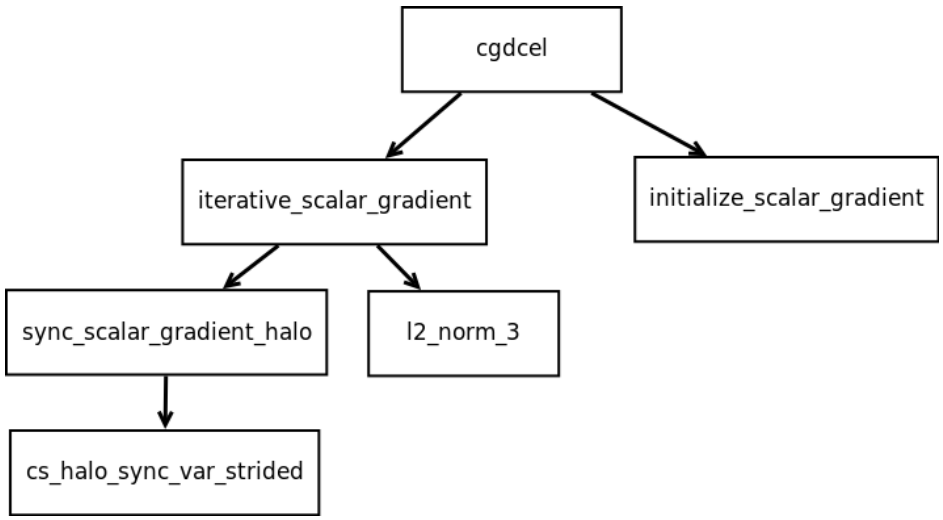


Figure 5.4: Call graph for the cgdcel_ function.

reslin_

This function computes the matrix-vector product that is part of the Jacobi iteration. The matrix-vector multiplication is also a two-step process. First up is the MPI communication and synchronization process executed in the cs_halo_sync_var function, followed by the matrix-vector computation performed in mat_vec_p_l_native. Figure 5.5 below shows the call graph for this function.

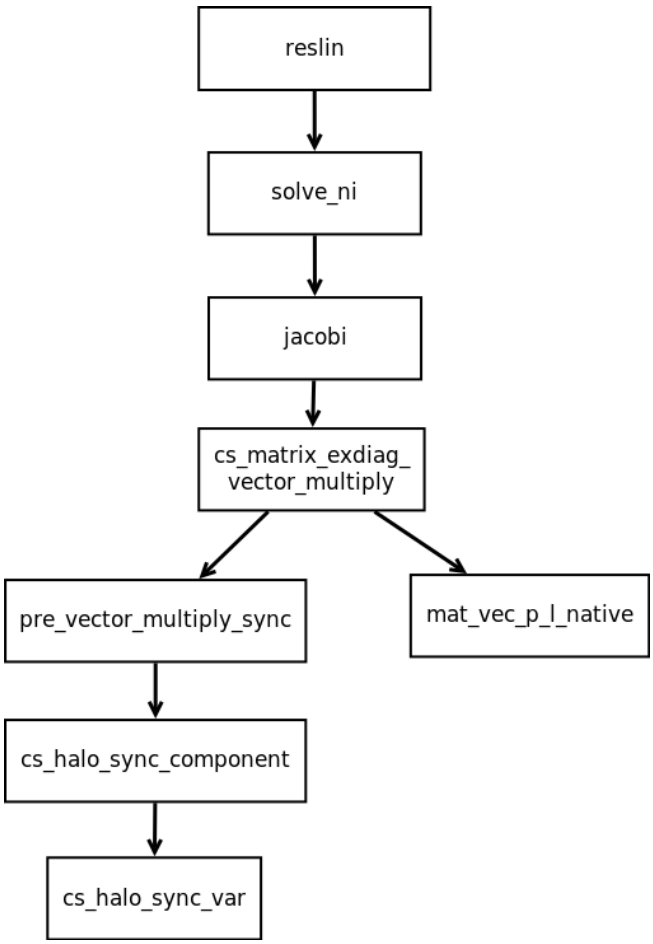


Figure 5.5: Call graph for the reslin_ function.

resmgr_

This function is the sparse linear system solver. It is responsible for around two thirds of the total run-time in our case. The multigrid solver is used as this is MPI communication is used. This function is divided in three sections: cs_sles_solve is the general sparse linear system solver. It makes use of three simple vector multiplication functions found in the cs_blas.c file. The second step is the matrix-vector multiplication, which is described in the reslin_ function above. The third step is the MPI communication and synchronization between each iteration. Figure 5.6 represents the call graph for this function.

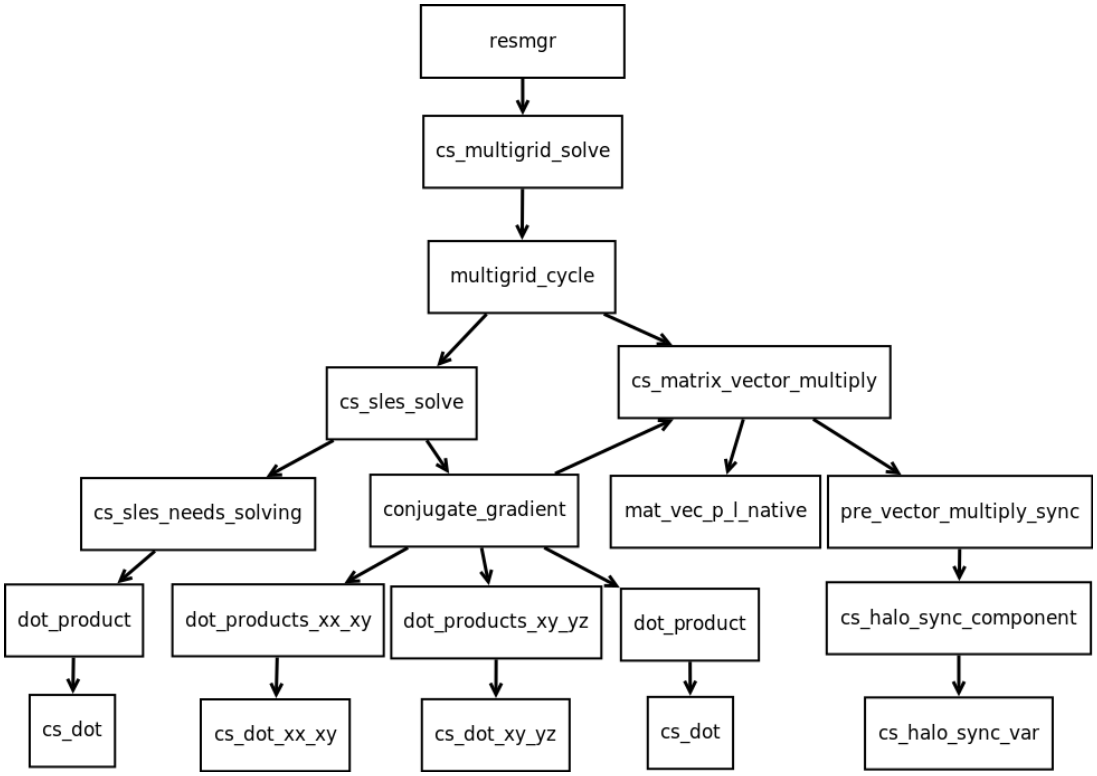


Figure 5.6: Call graph for the resmgr_ function.

5.3 Evaluation of base functions

This section will elaborate the base functions found in the profile analysis, explain their purpose and evaluate how well suited they are for an autotuning process.

cs_block_to_part_create_adj
This is a base function of the cs_preprocessor_data_read_mesh method. This function intializes the block to partition distributor for entities adjacent to already distributed entitites. The main work of this function is reading from file and data management. Additionally, this step is only executed once, meaning this function's contribution towards the total run-time diminishes as the number of iterations in the calculation increases. For these reasons it is not a good target for autotuning, and will not be autotuned.

bucket_sampling
This is another base function of the cs_preprocessor_data_read_mesh method. It is used to compute a sampling array which assumes a well-balanced distribution of leaves of the tree among the ranks. Like the function above, this one does not have any computationally intensive parts. And as a function part of preprocessing it will only be executed once. Thus function will not be autotuned as it would have no effect.

local_to_global_extents
The last base function of the cs_preprocessor_data_read_mesh method. This is a very small function, intended to transform local extents to global extents. The main body of this function is a single for loop, and could for that reason seem a viable target for optimization. However, it is responsible for only a few per cent of the total run-time. This ratio will only decrease as the number of iterations is increased. This function will not be autotuned.

fvm_box_tree_get_stats
The only base function of the cs_join_all function. It is responsible for gathering global box tree statistics. Its main task is data management of data organized in a tree structure, as well as printing statistics. Like the other three preprocessor functions listed above, it will not be autotuned.

l2_norm_3
This is a base function in the cgdcel method, and is thus part of the gradient calculation. More specifically, this function calculates the L^2 norm, or the square root of the sum of absolute values squared, of the input vector. The core of this function is computationally intensive, it is for that reason a possible autotuning target. But it is only responsible for a couple per cent of the total run-time. Any improvement from an autotuning process on this function would be neglibile, thus it will not be prioritized. To achieve better precision the l3superblock60 algorithm [32] is used.

cs_halo_sync_var & cs_halo_sync_var_strided
These two functions are handling the halo updates, that is, the border cells of each processors data set that must be shared between each iteration. This is a central task, so they are called from all of the computation methods. For this reason they are responsible for the majority of MPI-related run-time. Alas, autotuning have no effect on MPI functions, so these functions will not be optimized.

mat_vec_p_l_native
This is the first computationally intensive function that also contribute to a decent amount of the total run-time. The cs_matrix_exdiag_vector_multiply function is split in two parts, one synchronization step using the halo sync functions listed above. The other is the matrix-vector multiplication. This function is called from several core functions, and is one of the functions with the highest number of visits. There is a large variety of matrix-vector multiplication functions, which of them that is used is dependent on the matrix' properties. In our test case mat_vec_p_l_native is the used function. This function will be autotuned.

cs_dot
This function is found in the file cs_blas.c, which in itself is an indication that it involves mathematical calculations. This is a basic vector multiplication calculation, computing the product of vectors x and y, xy. It is called by several of the core functions, at various steps. To achieve better precision the l3superblock60 algorithm is also used in this function. This function is a prime target for autotuning, and will be an important test target.

cs_dot_xx_xy

This function is also found in the file cs_blas.c. It calculates two dot products of the two vectors x and y, returning x.x and x.y. These dot products could be computed separately using the function above, but simultaneous computing adds more optimization opportunities and cache behaviour, even before autotuning. The superblock algorithm is also used here. Just like the previous function this will be a prime target for an autotuning process.

cs_dot_xy_yz
This is the third function from the cs_blas.c file that is frequently called. It calculates two dot products of the three vectors x, y and z, returning x.y and y.z. Like the previous function, the two dot products are calculated simultanously for better performance, using the superblock algorithm for better precision. It is almost identical to the previous function, so also a good target for an autotuning process.

Chapter 6

Implementation

This chapter will give information about implementation considerations, and how the Orio directives were implemented for each of the selected functions. Orio will be used as an automatic performance tool in this project.

6.1 Implementation considerations

This section details the implementation process. Which autotuning parameters were chosen, and how encountered problems were solved.

6.1.1 Parameter space exploration strategies

The most detailed way to explore the given parameter space is by an exhaustive search. Though, this can often result in an infeasible procedure as the size of the search space can be exponentially large. As a consequence of this, alternative search heuristics are available.

Two effective, and more practical search heuristic strategies have been developed and integrated into Orio's search engine. These two heuristics are the Nelder-Mead Simplex method [33] and Simulated Annealing method [34]. The exhaustive approach is the default method in Orio, the preferred search method can however be specified by using the *search* argument. We used the exhaustive search, as feasible search durations were obtainable for the chosen search parameters and functions.

The space search can also be set to terminate early using the *time_limit* and *total_runs* arguments. If the search time exceeds the specified limit the search is suspended and code for the best optimization so far is returned. The number of runs enforces the search to finish in a specific quantity of full search moves. However, this can result in the returned configuration being far from the optimal, especially if limits are set considerably lower than they should be for the given parameter space.

To further improve the quality of the search result, each search heuristic is enhanced by applying a local search after the main search completes. This local search compares the best performance with neighboring coordinates. In the case that a better coordinate is discovered the local search will continue recursively until no further improvement is found.

6.1.2 Autotuning of problems with low workload

One problem encountered during the autotuning process was autotuning of problems with low workload. It was observed that if the execution time of a loop was around $1.0\text{x}10^{-8}$ or lower, Orio would evaluate the run-time as a flat 0. A specified amount of executions of the loop is done for every combination of autotuning parameters, and the run-time of all the tests are averaged. The resulting average becomes incorrect as these zero-values are included in the average.

This was observed during the autotuning of the Code_Saturne tutorial case, designed for use on a desktop computer. For this reason the autotuning of this case had to be discarded, since reasonable and useful results were unobtainable. This problem also impacted the tuning for the supercomputer-sized case. The superblock matrix-muliplication formula has a triple-nested for loop as the main body, which was autotuned, as well as a single for loop after the main body. This latter loop could not be reliably optimized for the tested cases as it was too small.

Results showed that in the case of simple vector addition or similar operations the size of the for loop had to be around 10000 for Orio to produce appreciable results. In the tutorial example designed for desktop PCs the size of this vector was never higher than 936.

6.1.3 Resolving various array size

One problem encountered during the autotuning process is that the size n of the arrays, or vectors, varies. As a result of this the for loops have various size. This size is one of, if not the most important consideration during the autotuning process. For this reason, a generalized autotuning for a specific n is likely to be very ineffective, even if autotuning is performed for the most important value of n.

This was resolved by printing all encountered values of n for a specific function to the output, and then parsing the output file summarizing the number of hits for each size n. That way autotuned functions could be generated for the values of n with the most hits, while the others would be directed to the original function. The code of the file parser is given in Appendix A.1.

6.1.4 Explanation of used parameter variables

The performance tuning process need a variety of configuration parameters. The used definitions and their parameters are listed below.

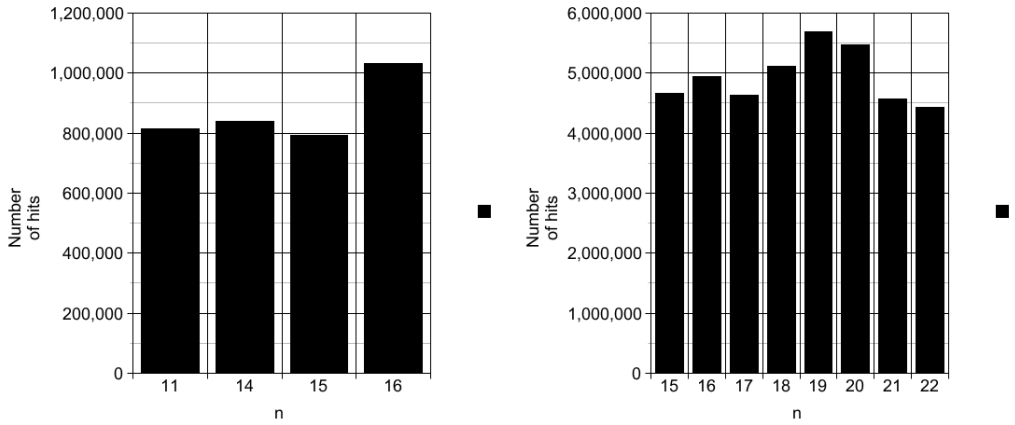
build - The *build_command* parameter is used to define the build command.
performance_counter - The *repetitions* parameter is used to define how many tests are run for each combination of tuning parameters.
performance_params - This section defines the tuning parameters. Their range or set of values is also defined here.
input_params - This section defiens the parameters used in the tuning process, such as loop size.
input_vars - This section defines all the variables used in the code segment that is to be autotuned. Initial values can also be set to zero, a specified or randomized value.
search - This section defines which the type of search that will be used, as well as parameters limiting the max length of the search.

6.2 Implementation of functions

This section will cover the implementation of the autotuning of all chosen functions. All implementations will be done for two configurations: one running on 1152 processors where MPI and computation are evenly distributed, and one on 288 processors where computation is responsible for the majority of the run-time. In the latter configuration it is easier to spot potential speedups related to computational optimizations.

6.2.1 mat_vec_p_l_native

The core of the computations in this function has source in a for loop setting up the matrix-vector multiplication. It can be found in Appendix A.2 The size n of the loop is a critical tuning parameter and must be obtained. Figures 6.1a and 6.1b shows the distribution of the loop size n for the two test cases.



(a) Distribution of loop sizes for 288 processors.

(b) Distribution of loop sizes for 1152 processors.

Figure 6.1: Distribution of loop sizes for the mat_vec_p_l_native function.

On 288 processors (left) there were around 114 million visits in total. For this case only values of n with more than 750000 hits are shown on the graph. When run on 1152 processors (right) there were around 424 million visits in total. In this case values of n with more than 4000000 hits are shown on the graph. That the distribution of visits is dependent on the number of processors is an important observation. This greatly degrades the chances of a flexible and automatic tuning process.

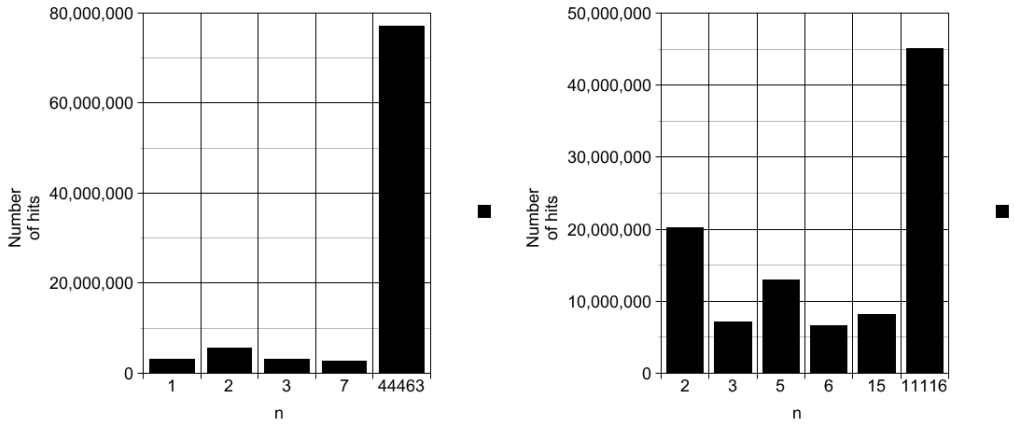
These are not good results. On 288 processors there is not a single value of n with more than 1% of total visits. As a result of this there is not really any values that can be prioritized for autotuning. It is only barely better when run on 1152. It is just as evenly distributed, and the value with the most visits has only just over 1% of the total visits.

For the autotuning to have any noticeable effect at least 20 functions would have to be autotuned. Considering each search can have a search time in hours, this is not a very feasible solution. Especially the flexibility of the solution is negatively impacted as these sizes and their distribution are dependent on both the input mesh(es) and the number of processors used. Even if the autotuning search times could be cut to half an hour it would still require 10 hours of autotuning to optimize 20% of this function only. It would also severely impact readability as twenty or more new and nearly similar functions would have to be implemented and called appropriately.

In addition it is the smallest loop sizes that have the most visits. As the work done in the loop decreases, in this case because of the loop size, the inaccuracy and inefficiency increases. Because of the even distribution of loop sizes and small workloads this function can not be autotuned efficiently.

6.2.2 cs_dot

This is the first of the three vector product functions. The function in its entirety can be found in Appendix A.3. The size n of the arrays, or vectors, must also here be extracted. The distribution of the size of n for the two configurations are given in Figures 6.2a and 6.2b below.



(a) Distribution of loop sizes for 288 processors.

(b) Distribution of loop sizes for 1152 processors.

Figure 6.2: Distribution of loop sizes for the cs_dot function.

On 288 processors (left) there were arround 77 million visits in total. For 288 processors only values of n higher than 2500000 are shown. When run on 1152 processors (right) there were nearly 260 million visits in total. In this case the threshold was set to 6000000 or higher. Once again the number of processors impacts the loop size.

This is much more promising results. For this function there is a single value of n much higher than the others, and it is also alone responsible for well over a third of the total visits. The autotuning process will be performed for these two values of n, 44463 and 11116 for 288 and 1152 processors respectively. The used tuning parameters, as well as the function that is to be autotuned is given in Figure 6.3 on the next page.

```
spec cs_dot_tune_spec {
def build {
 arg build_command = "gcc -fopenmp
        -D_POSIX_SOURCE -DHAVE_CONFIG_H
        -funsigned-char -pedantic -std=c99";
}
def performance_counter {
 arg repetitions = 50;
}
def performance_params {
 param Ui[] = range(1, 15);
 param Uj[] = range(1, 15);
 param Uk[] = range(1, 15);
}
def input_params {
 param n[] = [11116, 44463];
}
def input_vars {
 decl int sid = 0;
 decl int bid = 0;
 decl int i = 0;
 decl int start_id = 0;
 decl int end_id = 0;
 decl double sdot = 0.0;
 decl double cdot = 0.0;
 decl int block_size = 60;
 decl int n_blocks = n / block_size;
 decl int n_sblocks = sqrt(n_blocks);
 decl int blocks_in_sblocks =
   (n_sblocks > 0) ?
   n_blocks / n_sblocks : 0;
 decl double dot = 0.0;
 decl static double x[n] = random;
 decl static double y[n] = random;
}
def search {
 arg algorithm = "Exhaustive";
}}
```

```
/*@ begin PerfTuning (
 import spec cs_dot_tune_spec;
) @*/

/*@ begin Loop (
transform Unroll (ufactor=Ui)
for(sid = 0; sid <= n_sblocks-1; sid++){
 sdot = 0.0;

 transform Unroll (ufactor=Uj)
 for(bid = 0; bid <= blocks_in_sblocks-1; bid++){
  start_id = block_size *
               (blocks_in_sblocks*sid + bid);
  end_id = block_size *
               (blocks_in_sblocks*sid + bid + 1);
  cdot = 0.0;

  transform Unroll (ufactor=Uk)
  for (i = start_id; i <= end_id-1; i++)
   cdot += x[i]*y[i];
  sdot += cdot;
 }
 dot += sdot;
}
) @*/
/*@ end @*/
/*@ end @*/
```

Figure 6.3: The used tuning spec (left) and the code segment (right) that was autotuned.

This will generate two new tuned versions of the main for loop of the cs_dot function, one for each *n*. These new functions should only be called for their specific values of *n*. They are invoked from their parent function in the cs_sles.c file. Initially, this is done in one line:

```
double s = cs_dot(n_elts, x, y);
```

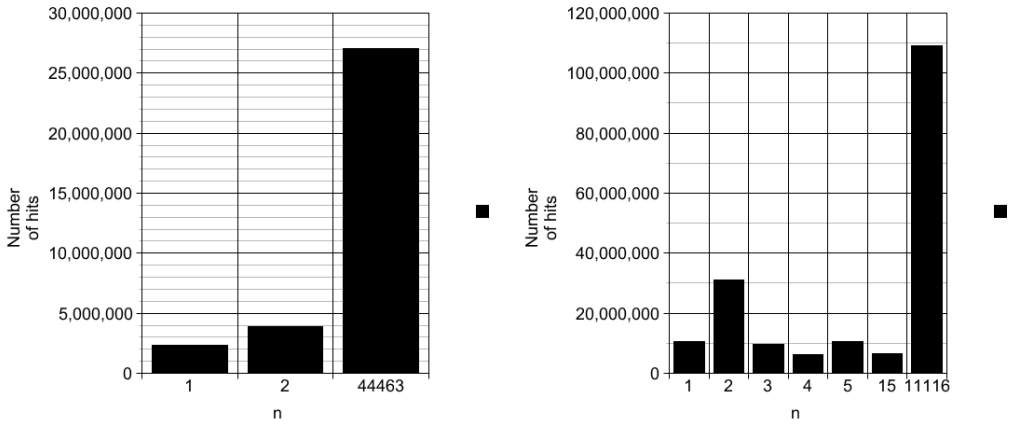
Where *n_elts* is an integer holding the array sizes. An if statement is created using this value to launch the appropriate function:

```
double s = 0.0;
if(n_elts == 11116){
  s = cs_dot_n_11116(n_elts, x, y);
}
else if(n_elts == 44463){
  s = cs_dot_n_44463(n_elts, x, y);
}
else
  s = cs_dot(n_elts, x, y);
```

The last step is to create the function definitons for these two new functions in the header file cs_blas.h. These are similar to the already existing definition of the cs_dot function, the function name is the only change.

6.2.3 cs_dot_xx_xy

This is the second of the three vector product functions. The full function can be found in Appendix A.4. Array size n of the arrays were obtained for both of the tests. The distribution of the size of n for the two configurations are given in Figures 6.4a and 6.4b below.



(a) Distribution of loop sizes for 288 processors.

(b) Distribution of loop sizes for 1152 processors.

Figure 6.4: Distribution of loop sizes for the cs_dot_xx_xy function.

On 288 processors (left) there were arround 77 million visits in total. For 288 processors only values of n higher than 2000000 are shown. When run on 1152 processors (right) there were just over 300 million visits in total. In this case the threshold was set to 6000000 or higher. Once again the number of processors impacts the loop size.

Once again good results. Just like *cs_dot* this function has a single value of n much higher than the others. It is the same values of n, also responsible for a third of the total visits. The autotuning process will be performed for these two values of n, 44463 and 11116 for 288 and 1152 processors respectively. The used tuning parameters, as well as the function that is to be autotuned is given in Figure 6.5 on the next page.

```
spec cs_dot_xx_xy_tune_spec {
def build {
 arg build_command = "gcc -fopenmp
        -D_POSIX_SOURCE -DHAVE_CONFIG_H
        -funsigned-char -pedantic -std=c99";
}
def performance_counter {
 arg repetitions = 50;
}
def performance_params {
 param Ui[] = range(1, 15);
 param Uj[] = range(1, 15);
 param Uk[] = range(1, 15);
}
def input_params {
 param n[] = [11116, 44463];
}
def input_vars {
 decl int sid = 0;
 decl int bid = 0;
 decl int i = 0;
 decl int start_id = 0;
 decl int end_id = 0;
 decl double sdot_xx = 0.0;
 decl double sdot_xy = 0.0;
 decl double cdot_xx = 0.0;
 decl double cdot_xy = 0.0;
 decl int block_size = 60;
 decl int n_blocks = n / block_size;
 decl int n_sblocks = sqrt(n_blocks);
 decl int blocks_in_sblocks =
        (n_sblocks > 0) ?
        n_blocks / n_sblocks : 0;
 decl double dot_xx = 0.0;
 decl double dot_xy = 0.0;
 decl static double x[n] = random;
 decl static double y[n] = random;
}
def search {
 arg algorithm = "Exhaustive";
}}
```

```
/*@ begin PerfTuning (
 import spec cs_dot_xx_xy_tune_spec;
) @*/

/*@ begin Loop (
transform Unroll (ufactor=Ui)
for (sid = 0; sid <= n_sblocks-1; sid++) {
 sdot_xx = 0.0;
 sdot_xy = 0.0;

 transform Unroll (ufactor=Uj)
 for (bid = 0; bid <= blocks_in_sblocks-1; bid++
  start_id = block_size *
                (blocks_in_sblocks*sid + bid);
  end_id = block_size *
       (blocks_in_sblocks*sid + bid + 1);
  cdot_xx = 0.0;
  cdot_xy = 0.0;

  transform Unroll (ufactor=Uk)
  for (i = start_id; i <= end_id-1; i++) {
   cdot_xx += x[i]*x[i];
   cdot_xy += x[i]*y[i];
  }
  sdot_xx += cdot_xx;
  sdot_xy += cdot_xy;
  }

  dot_xx += sdot_xx;
  dot_xy += sdot_xy;
}
) @*/

/*@ end @*/
/*@ end @*/
```

Figure 6.5: The used tuning spec (left) and the code segment (right) that was autotuned.

This will generate two new tuned versions of this code segment, one for each n. These new functions should only be called for their specific values of n. They are invoked from their parent function in the cs_sles.c file. Initially, this is done in one line:

```
cs_dot_xx_xy(n_elts, x, y, s, s+1);
```

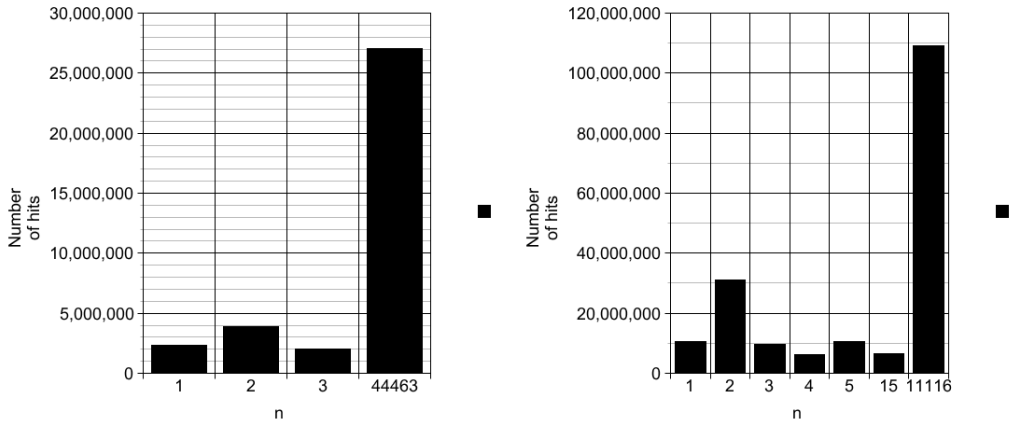
Where *n_elts* is an integer holding the array sizes. An if statement is created using this value to launch the appropriate function:

```
if(n_elts == 11116){
  cs_dot_xx_xy_n_11116(n_elts, x, y, s, s+1);
}
else if(n_elts == 44463){
  cs_dot_xx_xy_n_44463(n_elts, x, y, s, s+1);
}
else
  cs_dot_xx_xy(n_elts, x, y, s, s+1);
```

Unlike the cs_dot function this one does not return any values, but rather uses pointers to correctly place the computed values. Function definitions for these two new functions must be added in the header file cs_blas.h.

6.2.4 cs_dot_xy_yz

This is the final vector product function. The full function can be found in Appendix A.5 The sizes of n was extracted for both test cases. The distribution of the size of n for the two configurations are given in Figures 6.6a and 6.6b below.



(a) Distribution of loop sizes for 288 processors.

(b) Distribution of loop sizes for 1152 processors.

Figure 6.6: Distribution of loop sizes for the cs_dot_xy_yz function.

On 288 processors (left) there were arround 77 million visits in total. For 288 processors only values of n higher than 2000000 are shown. When run on 1152 processors (right) there were just over 300 million visits in total. In this case the threshold was set to 6000000 or higher. Once again the number of processors impacts the loop size.

Good results, as in the two previous functions. This function has a single value of n which is alone responsible for around a third of the total visits. The autotuning process will be performed for these two values of n, 44463 and 11116 for 288 and 1152 processors respectively. The used tuning parameters, as well as the function that is to be autotuned is given below in Figure 6.7.

```
spec cs_dot_xy_yz_tune_spec {
def build {
 arg build_command = "gcc -DHAVE_CONFIG_H
        -D_POSIX_SOURCE -DNDEBUG -O3 -std=c99
        -funsigned-char -pedantic -fopenmp";
}
def performance_counter {
 arg repetitions = 50;
}
def performance_params {
 param Ui[] = range(1, 15);
 param Uj[] = range(1, 15);
 param Uk[] = range(1, 15);
}
def input_params {
 param n[] = [11116, 44463];
}
def input_vars {
 decl int sid = 0;
 decl int bid = 0;
 decl int i = 0;
 decl int start_id = 0;
 decl int end_id = 0;
 decl double sdot_xy = 0.0;
 decl double sdot_yz = 0.0;
 decl double cdot_xy = 0.0;
 decl double cdot_yz = 0.0;
 decl int block_size = 60;
 decl int n_blocks = n / block_size;
 decl int n_sblocks = sqrt(n_blocks);
 decl int blocks_in_sblocks =
        (n_sblocks > 0) ?
        n_blocks / n_sblocks : 0;
 decl double dot_xy = 0.0;
 decl double dot_yz = 0.0;
 decl static double x[n] = random;
 decl static double y[n] = random;
 decl static double z[n] = random;
}
def search {
 arg algorithm = "Exhaustive";
}}
```

```
/*@ begin PerfTuning (
 import spec cs_dot_xy_yz_tune_spec;
) @*/

/*@ begin Loop (
transform Unroll (ufactor=Ui)
for (sid = 0; sid <= n_sblocks-1; sid++) {
 sdot_xy = 0.0;
 sdot_yz = 0.0;

 transform Unroll (ufactor=Uj)
 for (bid = 0; bid <= blocks_in_sblocks-1; bid++) {
  start_id = block_size *
        (blocks_in_sblocks*sid + bid);
  end_id = block_size *
        (blocks_in_sblocks*sid + bid + 1);
  cdot_xy = 0.0;
  cdot_yz = 0.0;

  transform Unroll (ufactor=Uk)
  for (i = start_id; i <= end_id-1; i++) {
   cdot_xy += x[i]*y[i];
   cdot_yz += y[i]*z[i];
  }
   sdot_xy += cdot_xy;
   sdot_yz += cdot_yz;
 }

 dot_xy += sdot_xy;
 dot_yz += sdot_yz;
}
) @*/
/*@ end @*/
/*@ end @*/
```

Figure 6.7: The used tuning spec (left) and the code segment (right) that was autotuned.

This will generate two new tuned versions of the main for loop of the cs_dot_xy_yz function, one for each n. These new functions should only be called for their specific values of n. Like the other two matrix multiplication functions they are called from the parent function in the cs_sles.c file. Initially, this is done in one line:

```
cs_dot_xy_yz(n_elts, x, y, z, s, s+1);
```

Where *n_elts* is an integer holding the array sizes. An if statement is created using this value to launch the appropriate function:

```
if(n_elts == 11116){
  cs_dot_xy_yz_n_11116(n_elts, x, y, z, s, s+1);
}
else if(n_elts = 44463){
  cs_dot_xy_yz_n_44463(n_elts, x, y, z, s, s+1);
}
else
  cs_dot_xy_yz(n_elts, x, y, z, s, s+1);
```

This functions is similar to the cs_dot_xy_yz function in that it uses pointers to correctly place the computed values. The function definitions for these newly created functions must be added in the header file cs_blas.h.

Chapter 7

Results and discussion

This chapter will present the results. First to be discussed is the inconsistent results encountered during the autotuning process. Next the results from running on 1152 processors will be presented. When running on this many processors Code_Saturene was was found to have a nice balance of run-time spent on computing and communication. The other set of tests were run on only 288 processors, resulting in a much bigger part of the run-time used on computation. This was done so potential speedups would be easier to detect.

7.1 Autotuning inconsistencies

As described in Section 6.1.2 autotuning of cases designed for desktop sized computers proved to be unreliable. Even when this was not the case, and the autotuning seemingly worked correctly the results from the autotuning process were not consistent. The chosen values for the unroll parameters did often vary from test to test, even if the code segment and the autotuning parameters were unchanged. As an example, for the *cs_dot* function the three unroll values from the outer to the inner loop were 4, 7 and 5 in the first search. In the second search unroll values of 9, 2 and 6 were reported as optimal values.

Though, no matter which unroll factors were reported to be the optimial configuration, they were all reported to have significantly shorter execution times than the original function with no unrolling. All of the functions were found to have unroll values offering between 20% and 30% speedup.

A higher number of test per value could resolve some of these problems, but that would greatly impact the execution time of each search. The implemented search configurations had a run-time of just under two hours with the 50 repetitions used per test. With 50 repetitions the average should be adequately accurate.

These results are ambiguous. On one hand they indicate there are possibilities for speedup, on the other hand the reliability must be questioned because of the inconsistencies.

7.2 Test results on 1152 processors

Running on 1152 processors offered a good balance between time spent on MPI communication and computations. This balance point is where the code has the lowest run-time, in other words the most likely configuration to be used. For that reason it is a good starting point for evaluating the effect of changes from the autotuning process.

The run-times were obtained from Code_Saturnes clocking values in the execution report. The values considered are the average CPU time for a single processor. Each test were run 10 times, every value is reported as well as the average. For the original code the results are depicted in Figure 7.1 on the next page. The average run-time was 436,954 seconds. The biggest deviations from the average were just over 5%.

In the next test Code_Saturne were compiled to use the ATLAS[35] libraries. The ATLAS project is an ongoing research effort attempting to provide a toolkit offering portable performance improvement with the use of empirical techniques. Figure 7.2 shows the results of these tests.

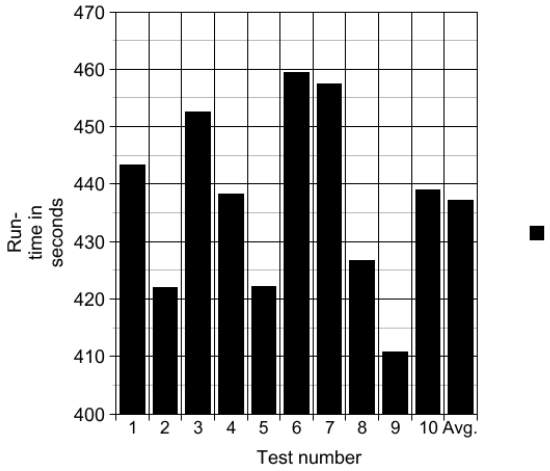


Figure 7.1: Test results for the original code on 1152 processors.

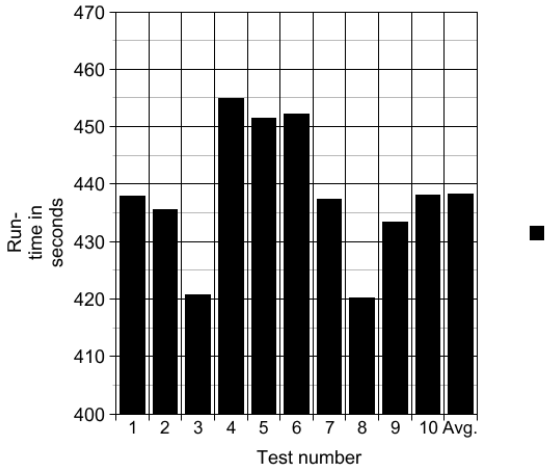


Figure 7.2: Test results for the original code compiled to use ATLAS on 1152 processors.

The average run-time when using ATLAS libraries was 438,190 seconds. With deviations in the same range, around 4%. In other words, there was no gain by using ATLAS libraries. The average run-time increased by quarter of a per cent, but considering the deviations this is most likely due to test deviation rather than slower algorithms.

The last tests were performed using the implemented modifications suggested by Orio's autotuning process. Three of the four computationally intensive functions were optimized, all three vector multiplication functions. While the matric-vector multiplication function could not be optimized, as explained in Section 6.2. Figure 7.3 shows the results for tests run on the autotuned version.

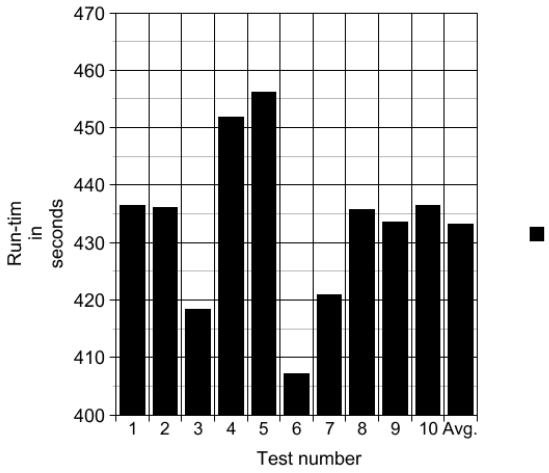


Figure 7.3: Test results for the autotuned code on 1152 processors.

The average run-time for the optimized code was 433,293 seconds. With deviations in the same range, around 4%. Compared to the 436,954 seconds for the original code the difference is neglible, this means autotuning was no improvement gained by autotuning the code. While there is a slight decrease in run-time this is, like the case for ATLAS, this is most likely due to test deviation rather than faster algorithms.

7.3 Test results on 288 processors

The other test configuration used only 288 processors during the execution phase. This leads to more time spent on computation than on MPI communication. This will result in a slower overall run-time, but it will also make changes in the average run-time more apparent. This test is intended to observe potential changes in the run-time that could have been overlooked in

the previous configuration.

For the sake of consistency the run-times were obtained from Code_Saturnes clocking values from the execution report in this test as well. The values considered are the average CPU time for a single processor. Each test were run 10 times, every value is reported as well as the average. For the original code the results are given below in Figure 7.4

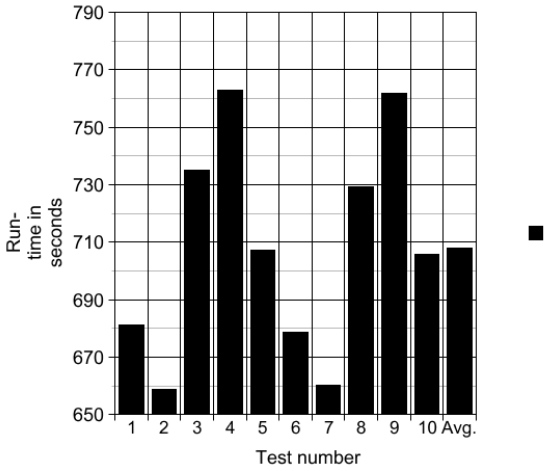


Figure 7.4: Test results for the original code on 288 processors.

With an average run-time of 708,072 seconds it is clear that this is a less optimal configuration than running on 1152 processors. Deviations also nearly doubled, now around 8%. This indicates that more time spent computing on each CPU leads to more inconsistent run-times.

The next test Code_Saturne on 288 processors used the ATLAS libraries. Figure 7.5 on the next page depicts the results of these tests. The average run-time using ATLAS on 288 processors was 719,820 seconds, once again a slight increase compared to the unmodified version. Still it is only a 1,5% increase, so the effect is neglible.

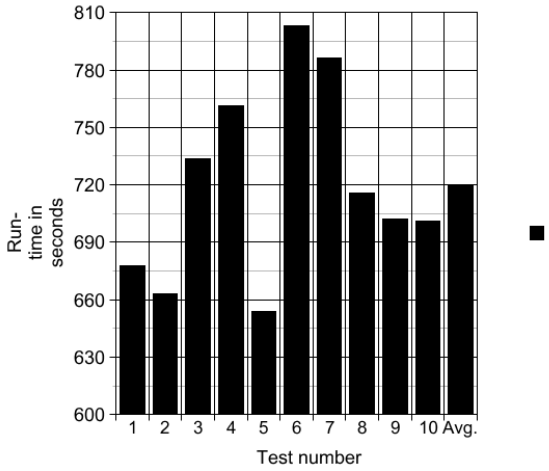


Figure 7.5: Test results for the original code compiled to use ATLAS on 288 processors.

The final test was performed used the implemented modifications from the autotuning process. With the exception of a reduced number of processors this test is identical to the one run on 1152 processors. Figure 7.6 shows the results for tests run on the autotuned version.

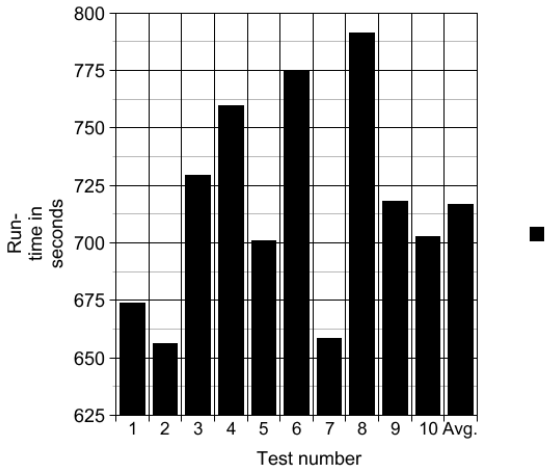


Figure 7.6: Test results for the autotuned code on 288 processors.

The average run-time in this test was 716,569 seconds. Deviations were the same as those in the two other tests on 288 processors. This average is slightly higher than the 708,072 seconds for the unmodified tests. In the balanced test autotuning was slightly quicker, in the case of more computation the impact should be more noticeable. However it was slightly slower instead.

This is a clear indication that the autotuning process, just like using the ATLAS libraries have no effect, and that the small differences are a result of deviation in tests. If more tests were performed for all the configurations the averages would have converged to the same value.

Chapter 8

Conclusions and future work

The full methodology of an autotuning process has been presented, from profiling analysis, through extracting important autotuning parameters to the actual implementation and testing the effect.

It has become apparent that the current design of Code_Saturne limits the possibilites for efficient autotuning. The length of the loops executed in the computationally heavy functions greatly varies. This loop length is the most important consideration in an autotuning process. An important observation is that this length is dependent on both the input mesh and the number of processors used, requiring code, execution parameters and input data to be adapted and tuned in conjunction. For some cases this distribution of loop sizes might be more beneficial with regards to autotuning, in others worse. This makes autotuning of Code_Saturne very unpredictable and inefficient.

The other limitation of the dependency between loop sizes and mesh size and number of processors is that any modification to either of these will require the full autotuning process to be redone for the new loop sizes. Cutting the number of processors used in half would require new empirical searches and implementations for each function. Even in cases where autotuning would offer great optimization, it would only be feasible to implement it if the cases executed many times with the same mesh and number of processors.

In the case tested the sizes of the loops were very evenly distributed: for the three vector multiplication functions it was only reasonable to autotune for a single value, responsible for a third of the sizes. In other words, only

a third of calls to these functions could be optimized for this specific case. Optimizing for additional loop sizes would greatly decrease readability, and also be very inefficient as the gain for each value would be neglible. The function handling matrix-vector multiplication could not be optimized at all as not a single loop size was used in more than 1% of the calls to that function. For this reason the majority of the computationally heavy functions were not fit for optimization with Orio, and no significant autotuning potential could be identified in the tested cases.
Orio did report potential for the autotuned for loops, suggesting that the run-times for the chosen code segments could be decreased by up to 30%. However, there were inconsistencies in the results of the autotuning process, as it did not suggest the same optimizations every time the same search was performed. No matter the suggested changes they were all reported to be quicker, but the reliability is questionable, since no effect was found from implementing the suggested modifications.

Another observation made was that Orio is not fit to tune loops with low workloads. If the time taken to execute the chosen code segment was too small Orio's test benchmarking was wrong. This was especially the case for workloads related to Code_Saturne problems designed for desktop computers, but it was also noticeable for the smaller loops in the case when running on a supercomputer.

The default code, the code compiled to use ATLAS and the autotuned code all had the same run-time, both when computation and MPI communication was balanced and when there was an imbalance towards more computation. The majority of the computationally heavy functions where not fit for improvements suggested by Orio, so they could only be implemented for a small portion of the code. Any potential effect was lost.

8.1 Future work

If the use of autotuning is to be further pursued large parts of Code_Saturne will have to be redesigned. If, *e.g.* the vector multiplication functions were called with the same and large array size almost every time the effect of autotuning would be greatly increased. However, this would require massive changes to the core of Code_Saturne, so whether it is a feasible option must

be considered.

Another option could be modifying data sets to result in more coherent subproblem sizes, resulting in greater potential for automatic tuning. Identifying important characteristic for such adaptions and applicable problems would make an interesting direction for further research.
Another option for further optimization is implement an option to use GPUs for the computationally intensive functions. This could greatly improve runtime, and it does not require large modifications, as it could be an additional option only requiring porting of a few core functions.

Bibliography

[1] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-Based Empirical Performance Tuning Using Orio," *Parallel & Distributed Processing*, pp. 1–11, May 2009.

[2] E. de France, "Code_Saturne practical user's guide."

[3] J. Owens, M. Houston, D. Luebke, and S. Green, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[4] M. Taher, "Accelerating Scientific Applications Using GPU's," *Design and Test Workshop (IDT)*, pp. 1–6, November 2009.

[5] R. Eidissen, "Utilizing GPUs for Real-Time Visualization of Snow," Master's thesis, Norwegian University of Science and Technology, February 2009.

[6] O. E. Krog and A. C. Elster, "Fast GPU-based Fluid Simulations Using SPH," *Lecture Notes in Computer Science*, vol. 7134, pp. 98–109, 2012.

[7] Électricité de France, "Code_Saturne."

[8] J. Nickolls and W. Dally, "The GPU Computing Era," *Micro, IEEE*, vol. 30, pp. 56–69, March 2010.

[9] T. Falch, J. Floystad, D. Breiby, and A. Elster, "Gpu-Accelerated Visualization of Scattered Point Data," *Access, IEEE*, vol. 1, pp. 564–576, 2013.

[10] P. Pacheco, *An Introduction to Parallel Programming*. Elsevier morgan kaufman, 2nd ed. ed., 2011.

[11] A. Aqrawi and A. Elster, "Bandwidth reduction through multithreaded compression of seismic images," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pp. 1730–1739, 2011.

[12] M. Flynn, "Some Computer Organizations and their Effectiveness," *Computers, IEEE Transactions on*, vol. C-21, pp. 948–960, September 1972.

[13] M. D. Hill, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, pp. 33–38, July 2008.

[14] J. L. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[15] J. C. Meyer, *Performance Modeling of Heterogeneous Systems*. PhD thesis, Norwegian University of Science and Technology, November 2012.

[16] J. Meyer and A. Elster, "Performance Modeling of Heterogeneous Systems," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–4, 2010.

[17] J. Carter, L. Oliker, and J. Shalf, "Performance Evaluation of Scientific Applications on Modern Parallel Vector Systems," *Proceedings of the 7th international conference on High performance computing for computational science*, pp. 490–503, 2006.

[18] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can Search Algorithms Save Large-Scale Automatic Performance Tuning?," *Proceedings ICCS 2011 in Procedia Computer Science*, November 2010.

[19] P. Balaprakash, S. M. Wild, and B. Norris, "Spapt: Search problems in automatic performance tuning," *Proceedings of the iCCS Workshop on Tools for Program Development and Analysis in Computational Science*.

[20] J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A Tool for top-down Analysis of Node Performance," *The Journal of Supercomputing*, vol. 23, pp. 81–104, August 2002.

[21] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parametrized Optimizations for Empirical Tuning," *Parallel and Distributed Processing Symposium*, pp. 1–8, March 2007.

[22] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris, "Autotuning Stencil-Based Computations on GPUs," *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pp. 266–274, September 2012.

[23] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An Auto-Tuning Framework for Parallel Multicore Stencil Computations," *2010 IEEE Symposium on Parallel & Distributed Processing*, pp. 1–12, April 2010.

[24] I.-H. Chung and J. K. Hollingsworth, "A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs," *Proceedings of the International Symposium on HIgh Performance Distributed Computing*, pp. 45–56, 2006.

[25] J. Forschungszentrum and G. R. S. for Simulation Sciences, "About Scalasca."

[26] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice & Experience - Scalable Tools for High-End Computing*, vol. 22, pp. 702–719, April 2010.

[27] Électricité de France, "Description of Code_Saturne."

[28] E. A. Fadlun, R. Verzicco, P. Orlandi, and J. Mohd-Yusof, *Combined Immersed-Boundary Finite-Difference Methods for Three-Dimensional Complex Flow Simulations*. 1997.

[29] E. A. Fadlun, R. Verzicco, P. Orlandi, and J. Mohd-Yusof, "Combined Immersed-Boundary Finite-Difference Methods for Three-Dimensional Complex Flow Simulations," *Journal of Computational Physics*, vol. 161, pp. 35–60, June 2000.

[30] NTNU, "About Vilje."

[31] SGI, "Message Passing Toolkit (MPT) User's Guide."

[32] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulus, "Reducing floating point error in dot product using the superblock family of algorithms," *SIAM J. SCI. COMPUT.*, vol. 31, no. 2, pp. 1156–1174, 2008.

[33] D. M. Olsson and L. S. Nelson, "The Nelder-Mead Simplex Procedure for Function Minimization," *Technometrics*, vol. 17, no. 1, pp. 45–51, 1975.

[34] S. Kirkpatrick, C. D. G. Jr, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671–680, May 1983.

[35] ATLAS, "Automatically Tuned Linear Algebra Software (ATLAS)."

Appendix A

Relevant code

A.1 parser

This is the parser file used to scan the output of the various array sizes.

```
#include <stdio.h>
#include <stdlib.h>

FILE *file;

void main()
{
  int i, temp;
  int lines = 0, threshold = 3000000;
  int array[50000];
  char line[10];

  for(i = 0; i < 50000; i++){
    array[i] = 0;
  }

  file = fopen("input.txt", "rt");
  while(fgets(line, 10, file) != NULL)
  {
sscanf (line, "%d", &temp);
    array[temp] += 1;
    lines += 1;
  }
  for(i = 0; i < 50000; i++){
    if(array[i] > threshold){
      printf("%d = %d\n", i, array[i]);
    }
  }
  printf("Lines parsed: %d\n", lines);
  fclose(file);
}
```

A.2 mat_vec_p_l_native

The is the matrix-vector multiplication function.

```
static void _mat_vec_p_l_native(bool                exclude_diag,
                    const cs_matrix_t  *matrix,
                    const cs_real_t    *restrict x,
                    cs_real_t          *restrict y)
{
  cs_lnum_t  ii, jj, face_id;
  const cs_matrix_struct_native_t  *ms = matrix->structure;
  const cs_matrix_coeff_native_t  *mc = matrix->coeffs;
  const cs_real_t  *restrict xa = mc->xa;

  /* Tell IBM compiler not to alias */
# if defined(__xlc__)
# pragma disjoint(*x, *y, *xa)
# endif

  /* Diagonal part of matrix.vector product */

  if (! exclude_diag) {
    _diag_vec_p_l(mc->da, x, y, ms->n_cells);
    _zero_range(y, ms->n_cells, ms->n_cells_ext);
  }
  else
    _zero_range(y, 0, ms->n_cells_ext);

  /* Note: parallel and periodic synchronization could be delayed to here */
  /* non-diagonal terms */

  if (mc->xa != NULL) {
    if (mc->symmetric) {
      const cs_lnum_t *restrict face_cel_p = ms->face_cell;

      for (face_id = 0; face_id < ms->n_faces; face_id++) {
        ii = face_cel_p[2*face_id] -1;
        jj = face_cel_p[2*face_id + 1] -1;
        y[ii] += xa[face_id] * x[jj];
        y[jj] += xa[face_id] * x[ii];
      }
    }
    else {
      const cs_lnum_t *restrict face_cel_p = ms->face_cell;
      for (face_id = 0; face_id < ms->n_faces; face_id++) {
        ii = face_cel_p[2*face_id] -1;
        jj = face_cel_p[2*face_id + 1] -1;
        y[ii] += xa[2*face_id] * x[jj];
        y[jj] += xa[2*face_id + 1] * x[ii];
      }
    }
  }
}
```

A.3 cs_dot

This is the first of the vector multiplication functions.

```
double cs_dot(cs_lnum_t  n,
       const cs_real_t  *x,
       const cs_real_t  *y)
{
  const cs_lnum_t block_size = 60;
  cs_lnum_t sid, bid, i;
  cs_lnum_t start_id, end_id;
  double sdot, cdot;
  cs_lnum_t n_blocks = n / block_size;
  cs_lnum_t n_sblocks = sqrt(n_blocks);
  cs_lnum_t blocks_in_sblocks = (n_sblocks > 0) ? n_blocks / n_sblocks : 0;
  double dot = 0.0;

# pragma omp parallel for reduction(+:dot) private(bid, start_id, end_id, i, \
                                                   cdot, sdot) if (n > THR_MIN)
  for (sid = 0; sid < n_sblocks; sid++) {
    sdot = 0.0;

    for (bid = 0; bid < blocks_in_sblocks; bid++) {
      start_id = block_size * (blocks_in_sblocks*sid + bid);
      end_id = block_size * (blocks_in_sblocks*sid + bid + 1);
      cdot = 0.0;
      for (i = start_id; i < end_id; i++)
        cdot += x[i]*y[i];
      sdot += cdot;
    }
    dot += sdot;
  }
  cdot = 0.0;
  start_id = block_size * n_sblocks*blocks_in_sblocks;
  end_id = n;
  for (i = start_id; i < end_id; i++)
    cdot += x[i]*y[i];
  dot += cdot;

  return dot;
}
```

A.4 cs_dot_xx_xy

This is the second vector multiplication function, working on three vector.

```
void cs_dot_xx_xy(cs_lnum_t               n,
             const cs_real_t  *restrict   x,
             const cs_real_t  *restrict   y,
             double                      *xx,
             double                      *xy)
{
  const cs_lnum_t block_size = 60;
  cs_lnum_t sid, bid, i;
  cs_lnum_t start_id, end_id;
  double sdot_xx, sdot_xy, cdot_xx, cdot_xy;
  cs_lnum_t n_blocks = n / block_size;
  cs_lnum_t n_sblocks = sqrt(n_blocks);
  cs_lnum_t blocks_in_sblocks = (n_sblocks > 0) ? n_blocks / n_sblocks : 0;
  double dot_xx = 0.0;
  double dot_xy = 0.0;
#if defined(__xlc__)
#pragma disjoint(*x, *y, *xx, *xy)
#endif

# pragma omp parallel for private(bid, start_id, end_id, i, cdot_xx, cdot_xy, sdot_xx, sdot_xy) \
                          reduction(+:dot_xx, dot_xy) if (n > THR_MIN)
  for (sid = 0; sid < n_sblocks; sid++) {
    sdot_xx = 0.0;
    sdot_xy = 0.0;
    for (bid = 0; bid < blocks_in_sblocks; bid++) {
      start_id = block_size * (blocks_in_sblocks*sid + bid);
      end_id = block_size * (blocks_in_sblocks*sid + bid + 1);
      cdot_xx = 0.0;
      cdot_xy = 0.0;
      for (i = start_id; i < end_id; i++) {
        cdot_xx += x[i]*x[i];
        cdot_xy += x[i]*y[i];
      }
      sdot_xx += cdot_xx;
      sdot_xy += cdot_xy;
    }
    dot_xx += sdot_xx;
    dot_xy += sdot_xy;
  }
  cdot_xx = 0.0;
  cdot_xy = 0.0;
  start_id = block_size * n_sblocks*blocks_in_sblocks;
  end_id = n;
  for (i = start_id; i < end_id; i++) {
    cdot_xx += x[i]*x[i];
    cdot_xy += x[i]*y[i];
  }
  dot_xx += cdot_xx;
  dot_xy += cdot_xy;

  *xx = dot_xx;
  *xy = dot_xy;
}
```

A.5 cs_dot_xy_yz

This is third and last vector multiplication function, also working on three vectors.

```
void cs_dot_xy_yz(cs_lnum_t               n,
             const cs_real_t  *restrict   x,
             const cs_real_t  *restrict   y,
             const cs_real_t  *restrict   z,
             double                      *xy,
             double                      *yz)
{
  const cs_lnum_t block_size = 60;
  cs_lnum_t sid, bid, i;
  cs_lnum_t start_id, end_id;
  double sdot_xy, sdot_yz, cdot_xy, cdot_yz;
  cs_lnum_t n_blocks = n / block_size;
  cs_lnum_t n_sblocks = sqrt(n_blocks);
  cs_lnum_t blocks_in_sblocks = (n_sblocks > 0) ? n_blocks / n_sblocks : 0;
  double dot_xy = 0.0;
  double dot_yz = 0.0;
#if defined(__xlc__)
#pragma disjoint(*x, *y, *xy, *yz)
#endif

# pragma omp parallel for private(bid, start_id, end_id, i, cdot_xy, cdot_yz, sdot_xy, sdot_yz) \
                                  reduction(+:dot_xy, dot_yz) if (n > THR_MIN)
  for (sid = 0; sid < n_sblocks; sid++) {
    sdot_xy = 0.0;
    sdot_yz = 0.0;
    for (bid = 0; bid < blocks_in_sblocks; bid++) {
      start_id = block_size * (blocks_in_sblocks*sid + bid);
      end_id = block_size * (blocks_in_sblocks*sid + bid + 1);
      cdot_xy = 0.0;
      cdot_yz = 0.0;
      for (i = start_id; i < end_id; i++) {
        cdot_xy += x[i]*y[i];
        cdot_yz += y[i]*z[i];
      }
      sdot_xy += cdot_xy;
      sdot_yz += cdot_yz;
    }
    dot_xy += sdot_xy;
    dot_yz += sdot_yz;
  }
  cdot_xy = 0.0;
  cdot_yz = 0.0;
  start_id = block_size * n_sblocks*blocks_in_sblocks;
  end_id = n;
  for (i = start_id; i < end_id; i++) {
    cdot_xy += x[i]*y[i];
    cdot_yz += y[i]*z[i];
  }
  dot_xy += cdot_xy;
  dot_yz += cdot_yz;
  *xy = dot_xy;
  *yz = dot_yz;
}
```